More Recursion

FRIDAY, APRIL 18

Continuing our exploration of recursion by discussing Binary Search, then generating sequences and permutations.

- 📚 Readings: Text 8.1, 8.2, 8.3
- 📝 Lecture quiz on Canvas
- 🎬 Lecture video on Canvas

Lecture Video

Click to sign in and play video

Prezis

For ease of access, here are the coin flip and permutations Prezis from today's lecture:

Recursion - Coin Flip

SS **By Sean Szumlanski**
April 16, 2025

Present

Recursion - Permutations v1 (soFar + rest)

SS By Sean Szumlanski
April 19, 2025

Present

Contents

Linear Search

We dove straight into some code today. We started by writing a function that takes a vector of integers and some integer we want to search for, `key` , and returns `true` if the vector contains the key, `false` otherwise. This function simply checks the vector sequentially, starting at index 0, and stops as soon as it finds the key. This particular approach to searching is called "linear search." Here it is in code:

```
// Takes a vector of integers and some key to search for. Returns true if key is
// in the vector, false otherwise. Passing vector by reference for efficiency.
bool search(Vector<int>& v, int key)
{
   for (int i = 0; i < v.size(); i++)
   {
      if (v[i] == key)
      {
         return true;
      }
   }

   // If we make it through the loop without finding our key, it must not be in
   // our vector.
   return false;
}
```

A common alternative approach to a search function like this is to return the first index where we find the key we're searching for. In that case, we often return -1 to indicate failure to find the key, because -1 is never a valid index in a vector.

```
// Takes a vector of integers and some key to search for. Returns the first index
// where key occurs, or -1 if key is not present. Passing vector by reference for
// efficiency.
int search(Vector<int>& v, int key)
{
   for (int i = 0; i < v.size(); i++)
   {
      if (v[i] == key)
      {
         return i;
      }
   }

   // If we make it through the loop without finding our key, it must not be in
   // our vector.
   return -1;
}
```

Best- and Worst-Case Runtime

The linear search function is unlike any of the other functions we analyzed in our Big-O lecture because its runtime isn't always a function of *n*. If *n* is arbitrarily large, but the key we're searching for is at index 0 in the vector, we end up with what looks like an O(1) ("constant") runtime. However, if the key is not in the vector, we end up searching the entire thing, which takes O(n) ("linear") time. (Recall that passing a vector to a function by reference is an O(1) operation. Passing by value would be O(n) because we would have to make a copy of the vector. Recall also that each call to `v.size()` is O(1).)

This led to our discussion of **best-** and **worst-case runtime**. We say the best-case runtime for linear search is O(1), and the worst-case runtime is O(n).

(*Key take-away!*) (***Insanely important!***) When discussing best- and worst-case runtimes, we *always* assume that the inputs to our function (in this case, the size of the vector) are arbitrarily **HuGe**! So, we never say the best-case runtime corresponds to the situation where the vector has just one element (or no elements). If we did that, then the discussion of best-case runtime for *every* function would be come moot. We would just never pass them any inputs, and finish in constant time without even executing them. Furthermore, if we were to restrict our consideration to inputs of size *n* = 1, then saying our runtime was O(1) would necessarily be the same as saying our runtime was O(n) (since 1 and *n* are equal in that case), and that's just not correct.

(*Key take-away!*) Generally speaking, if someone asks for the runtime of some function, we are implicitly looking for **worst-case runtime** unless otherwise specified.

Generating Vectors of Random Integers

To test our linear search function, we needed to generate vectors of random integers. To do that, I capitalized on the `randomInteger(low, high)` function from Stanford's random.h library. That function takes two integer parameters, `low` and `high`, and returns an integer on the range `low` through `high` (inclusive).

(*Only mentioned in passing in class: note about CS vernacular.*) My use of the word "inclusive" above in the phrase "`low` through `high` (inclusive)" means that the function has the ability to return `high` as one of its values. Sometimes we want to indicate that the range of a function's return value range goes all the way up to, but does not include, `high`. In that case, we would say the function returned an integer on the range "`low` to `high` (exclusive)." The "exclusive" here generally refers to the highest endpoint and not the lowest endpoint of the range; this phrasing generally indicates that `low` could be returned, but not `high`. If neither endpoint were returnable, we would likely say the function returned an integer between `low` and `high` (exclusive). (Notice that when distinguishing between `high` being included or not, I also personally tend to switch prepositions: "through" for inclusive, "to" for exclusive.)

At any rate, we used `randomInteger(low, high)` to write a function that created vectors of *n* random integers, and used those to test our search function. The resulting code is as follows:

```
#include <iostream>
#include "console.h"
#include "random.h"  // for randomInteger()
#include "simpio.h"  // for getInteger()
#include "vector.h"
using namespace std;


// Creates and returns a vector of n random integers on the range 0 through
// 100 (inclusive). Assumes n is non-negative.
Vector<int> createRandoVector(int n)
{
   Vector<int> v;

   for (int i = 0; i < n; i++)
   {
      // Generate random integer on the range 0 through 100 (inclusive) and add
      // to vector.
      v.add(randomInteger(0, 100));
   }

   return v;
}

// Takes a vector of integers and some key to search for. Returns the first index
// where key occurs, or -1 if key is not present. Passing vector by reference for
// efficiency.
int search(Vector<int>& v, int key)
{
   for (int i = 0; i < v.size(); i++)
   {
      if (v[i] == key)
      {
         return i;
      }
   }

   // If we make it through the loop without finding our key, it must not be in
   // our vector.
   return -1;
}

int main()
{
   // Create vector with 10 random integers and print to screen.
   Vector<int> v = createRandoVector(10);
   cout << v << endl;

   while (true)
   {
      cout << endl << "What integer shall we search for? (-1 to quit) ";
      int key = getInteger();

      if (key == -1)
      {
         // Jump to end of loop and continues executing main().
         break;
      }

      cout << "result: " << search(v, key) << endl;
   }

   cout << "Bye!" << endl;
   return 0;
}
```

Generating Sorted Vectors (and a Note About the Word "Sorted")

After testing our linear search function, I mentioned that it would work whether the vector we passed to it was sorted or not. I then took a moment to talk about two approaches for generating sorted vectors of random integers. Note that when I use the word "sorted" this quarter, I implicitly mean sorted from least to greatest or, more precisely, sorted in **non-decreasing order**. (This is not quite the same as saying that something is sorted in increasing order. Do you remember the distinction?)

The first approach simply relies on the built in `sort()` function for vectors. We generate a vector as above, but sort just before returning:

```
// Creates and returns a vector of n random integers on the range 0 through
// 100 (inclusive). Assumes n is non-negative.
Vector<int> createSortedRandoVector(int n)
{
   Vector<int> v;

   for (int i = 0; i < n; i++)
   {
      v.add(randomInteger(0, 100));
   }

   v.sort();
   return v;
}
```

(*Not mentioned in class.*) Alternatively, we could generate a sorted and random(ish) vector without calling `sort()` at all, using the following approach:

```
// Creates and returns a vector of n random integers on the range 0 through
// 100 (inclusive). Assumes n is positive.
Vector<int> createSortedRandoVector(int n)
{
   Vector<int> v;

   // This assumes n > 0, and so we actually want to add at least one element to
   // our vector. We could check manually whether n were <= 0 (and nope out of this
   // function if so), but I am just assuming n > 0 here for simplicity.
   v.add(randomInteger(0, 10));

   for (int i = 1; i < n; i++)
   {
      // By adding a non-negative integer to the value stored in the previous cell,
      // we ensure this cell gets a value greater than or equal to the previous one.
      // Thus, our vector ends up sorted.
      v.add(v[i - 1] + randomInteger(0, 10));
   }

   return v;
}
```

Note that the latter approach throws off the probability distribution and is not recommended in applications where a uniform distribution is desired. This is presented strictly to show an interesting approach to getting a sorted and random(ish) list of integers in the event that we don't have a sort function available to us. (Historical side note: I used to use this approach to generate random(ish) sorted containers in lecture when teaching classes in C (not C++). C's built-in sorting function isn't the most friendly, and I often didn't have time to clearly explain the details of how it worked in class, so I instead used this other approach as a stand-in whenever it was good enough for our purposes at the time.)

Binary Search

Having created sorted vector, we next discussed **binary search** -- a search algorithm that capitalizes on the sortedness of our vectors to give us faster runtimes than linear search. The main idea behind binary search is that we start by examining the element in the **middle** of our vector. If the key we're searching for is less than that middle element, then if it appears in the vector at all, it must be to the *left* of that middle element (since our vector is sorted). That means we can eliminate the ***entire right half*** of the vector from our search space! Similarly, if the key we're searching for is *greater* than the middle element, it must be to the *right* of that middle element, which allows us to eliminate the ***entire left half*** of the vector from our search space!

From there, we repeatedly go to the middle element of our remaining search space until we either find the key or whittle our search space down to zero elements. A more detailed explanation of binary search is available in today's lecture at timestamp 22:00. An explanation of the recursive implementation of binary search is available at timestamp 33:00. Here is our resulting code:

```
#include <iostream>
#include "console.h"
#include "random.h"
#include "simpio.h"
#include "vector.h"
using namespace std;

// Binary search from positions lo through hi (inclusive) for our key. Return first
// index where found, or -1 if not found. Passing vector by reference for efficiency.
int binarySearch(Vector<int>& v, int key, int lo, int hi)
{
   // Base case: If our indices have crossed over, the key must not be present.
   if (lo > hi)
   {
      return -1;
   }

   // See note about this midpoint formula in the section of today's notes
   // titled, "Midpoint Formula (and Integer Overflow)."
   int mid = lo + (hi - lo) / 2;

   if (key < v[mid])
   {
      return binarySearch(v, key, lo, mid - 1);
   }
   else if (key > v[mid])
   {
      return binarySearch(v, key, mid + 1, hi);
   }
   else
   {
      // Hooray, we found it. :)
      return mid;
   }
}

// Wrapper function for binary search. This is the one our client would call.
int binarySearch(Vector<int>& v, int key)
{
   return binarySearch(v, key, 0, v.size() - 1);
}

Vector<int> createSortedRandoVector(int n)
{
   Vector<int> v;

   for (int i = 0; i < n; i++)
   {
      v.add(randomInteger(0, 50));
   }

   v.sort();
   return v;
}

int main()
{
   Vector<int> v = createSortedRandoVector(10);
   cout << v << endl;

   while (true)
   {
      cout << endl << "What integer shall we search for? (-1 to quit) ";
      int key = getInteger();

      if (key == -1)
      {
         break;
      }

      cout << "-> result: " << binarySearch(v, key) << endl;
   }

   cout << "Bye!" << endl;
   return 0;
}
```

Binary Search Runtime (and a Reminder of the Awesomeness of Logarithmic Runtimes!)

Notice that with each O(1) comparison to the midpoint of our remaining search space, binary search cuts our search space in half. This repeated halving of our search space matches the pattern we saw for logarithmic runtimes in our lecture on Big-O. Indeed, the worst-case runtime for binary search is O(log n). The best-case runtime is O(1), where the key we're searching for is at the very middle of the vector and is therefore the first element we examine.

Recall from our intro lecture on Big-O that logarithmic runtimes are ***AMAZiNG***! I mentioned that day that $\log_2$(1 billion) ≈ 30. That means that if we have a sorted vector, binary search can determine whether it contains a particular element with just ***30 operations at most!!!!*** (Give or take a multiplicative constant, of course. This is Big-O, after all.) Compare that to linear search, which would potentially have to examine ***all 1 billion elements*** to determine whether a particular key was present in the vector. This is a ***vast*** disparity whose amazingness cannot be overstated.

Supplementary Midpoint Formula (and Integer Overflow)

On most systems, an `int` variable in C++ can hold values -2,147,483,648 through 2,147,483,647. If we add 1 to the maximum possible integer value, it actually wraps back around to the opposite end of the range and gives us -2,147,483,648. This is called **integer overflow**:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
   int biggest = 2147483647;
   biggest++;

   cout << biggest << endl;  // prints -2147483648

   return 0;
}
```

Here's why this is relevant to today's lecture: in the binary search function above, we might be tempted to stick with the following formula to calculate the midpoint between `lo` and `hi` :

```
int mid = (lo + hi) / 2;
```

Consider what happens if we end up binary searching a vector with two billion and one elements, though:

- Initially, lo = 0 and hi = 2,000,000,001. The midpoint formula above gives us mid = 1,000,000,000 (after integer truncation).
- Now suppose the key we're searching for is greater than the element at index 1,000,000,000. In that case, we set lo = mid + 1. hi is unchanged.
- We now have lo = 1,000,000,001 and hi = 2,000,000,001. The midpoint formula above gives us (1,000,000,001 + 2,000,000,001) / 2. In the real world, that's all fine and dandy, but in C++, that addition operation gives us integer overflow; C++ can't handle the value 3,000,000,002 when dealing with integers. In this case, integer overflow gives us mid = (lo + hi) / 2 = -1,294,967,294 / 2 = −647,483,647, which is an invalid index. Our program would crash spectacularly if we tried to access that position in a vector.

The alternative formula, `mid = lo + (hi - lo) / 2;` , is algebraically equivalent to the one above, but as long as `lo` and `hi` are valid integers to begin with, this one cannot possible result in integer overflow. (Trace through that formula with `lo = 1,000,000,001` and `hi = 2,000,000,001` to convince yourself that we never encounter an intermediary result that is greater than `hi` , and therefore, if `hi` itself was a valid non-negative integer, we cannot possibly encounter integer overflow with this formula.)

This might all seem a bit pedantic, but it has been known to come up in tech interviews.

Function Overloading

When we talked about wrapper functions on Wednesday, I used the following naming convention (approximately):

```
void buzzyHelper(...)
{
   ...
}

void buzzy(...)
{
   buzzyHelper(...);
}
```

When writing our `binarySearch()` function today, instead of appending `Helper` or `ForRealsies` to the end of the name of our helper function, I simply gave both the wrapper and recursive helper functions the same name:

```
int binarySearch(Vector<int>& v, int key, int lo, int hi)
{
   ...
}

int binarySearch(Vector<int>& v, int key)
{
   return binarySearch(v, key, 0, v.size() - 1);
}
```

This is an example of **function overloading** in C++. Function overloading is where we give multiple functions the same name and allow C++ to determine which one to call based on the number and types of parameters we're passing to the function. This streamlines our code somewhat by freeing us from the need to clutter and elongate the names of our helper functions.

Recursive Sequence Generation: Coin Flips

For this section, see related Prezi at the top of today's notes.

We then examined the recursive function I had alluded to (briefly) in our Big-O lecture for generating all possible sequences that can come from flipping a coin some number of times. For example, if we flip a coin 3 times, the sequences we might encounter are as follows (where 'H' represents heads, 'T' represents tails):

- HHH
- HHT
- HTH
- HTT
- THH
- THT
- TTH
- TTT

For an explanation of how to generate these coin-flip sequences, as well as a walk-through of the following code, see timestamp 39:40 of today's lecture. In the Prezi, we walked through a **recursive tree diagram** (sometimes referred to as a **decision tree** in problems like this, where each recursive call involves making some sort of choice -- such as whether to flip a head or a tail). Hopefully the diagram helped shed some light on how the recursive algorithm works and how each of these sequences is generated.

Following is the coin flip code (peppered with explanatory comments), which is very short, yet immensely powerful. It's generating an explosion of sequences -- 2^n of them, to be exact:

```
#include <iostream>
#include "console.h"
using namespace std;

// Prints all possible coin flip sequences (consisting of characters 'H' and/or 'T')
// starting with the sequence we have generated so far (soFar) and ending with n
// additional coin flips.
void coinFlip(string soFar, int n)
{
   // If there are no more coins to flip, we have generated a sequence of the
   // desired length. Print it.
   if (n == 0)
   {
      cout << soFar << endl;
      return;
   }

   // If there are coins remaining to be flipped, generate both possibilities:
   // one where we flip a head (adding 'H' to the sequence we've generated so far)
   // and one where we flip a tail (adding 'T' to soFar). In both cases, we have
   // one less coin to flip. We make both recursive calls because we want to
   // generate all possible sequences, which necessarily means exploring both
   // possible outcomes for this particular coin flip.
   coinFlip(soFar + "H", n - 1);
   coinFlip(soFar + "T", n - 1);
}

// Our wrapper function, which serves as a gateway to the recursive coinFlip()
// function above. Generates all possible coin flip sequences (made up of the
// characters 'H' and 'T') of length n.
void coinFlip(int n)
{
   // The empty string we're passing as our first parameter below indicates that
   // we are starting with an empty sequence. We have not yet flipped any coins,
   // and so the sequence does not yet contain any 'H' or 'T' characters.
   coinFlip("", n);
}

int main()
{
   coinFlip(3);
   return 0;
}
```

output:

```
HHH
HHT
HTH
HTT
THH
THT
TTH
TTT
```

Recursive Permutation Generation

For this section, see related Prezi at the top of today's notes.

The final recursive function we examined today generated all permutations of a given string. A **permutation** is just a re-ordering of elements. For example, each of the following strings is a permutation of the characters in "cat":

- cat
- cta
- act
- atc
- tac
- tca

The approach we saw today generated a `soFar` string that kept track of the permutation we had built so far on our recursive journey through the function. It also maintained a `rest` string with the characters that had yet to be included in `soFar` at each step of the way. For an explanation of this particular approach to generating permutations, as well as a walk-through of the following code, see timestamp 47:45 of today's lecture. Our final approach to generating permutations was as follows:

```
#include <iostream>
#include "console.h"
using namespace std;

void permute(string soFar, string rest)
{
   if (rest == "")
   {
      cout << soFar << endl;
      return;
   }

   for (int i = 0; i < rest.length(); i++)
   {
      // Generate a new copy of the "rest" string with the character
      // at index i removed.
      string newRest = rest.substr(0, i) + rest.substr(i + 1);
      permute(soFar + rest[i], newRest);
   }
}

void permute(string s)
{
   permute("", s);
}

int main()
{
   permute("act");
   return 0;
}
```

output:

```
cat
cta
act
atc
tac
tca
```

What's next?

We will spend the next three days of class talking about recursion. On Monday, we will talk about fractals and see a few other examples of recursive functions related to today's sequences and permutations functions. On Wednesday and Friday of next week, we will talk about recursive backtracking.

Practice Exercises

1. Be sure to check out the **Supplementary** section of today's notes that talks about integer overflow. That particular problem sometimes comes up in technical interviews.

2. After reviewing today's notes, be sure to take a break and then code up all three recursive functions from class today from scratch, without referring back to the notes. This will help you hone yours skills with "safe" problems whose solutions you have already seen, which can serve as a gentle stepping stone to the section problems and assignment problems if you're finding some of those a bit too difficult to jump into straight away.

3. Be sure to review the permutations Prezi and poke at the code to see how that function works. This is a fairly sophisticated function, and it was covered quite quickly today. Trace through the code either by drawing a decision tree diagram by hand, following along with the Prezi for this problem, or using the debugger to step through the code and watch the strings change with each recursive call.

4. When coding binary search today, we saw two approaches to calculating the midpoint between two integers. Which of those approaches was capable of producing incorrect results, and why was that the case? Also, prove that both approaches are algebraically equivalent, despite the fact that one of them could, in practice, introduce a bug into our binary search function.

5. Write an iterative version of binary search. Test it carefully on sorted arrays of varying lengths and compositions.

6. What is the Big-O runtime for the `coinFlip()` function from today's lecture?

Highlight for solution to Problem #6: This is a bit tricky. Since each recursive call spawns to more recursive calls until we hit our base cases, it might look like this function's runtime is O(2^n) ("exponential"). It's actually worse than that, though, because we are passing strings by value -- meaning that we create an entirely new copy of our string with each recursive call -- and those strings keep getting longer and longer. The details of the analysis for this function are beyond the scope of anything we have covered so far in class, and so you actually needn't worry about the answer to this problem, but the runtime ends up being $O(n2^n)$. I am bringing this up mostly to ensure that if anyone was looking at that function today and thinking it was $O(2^n)$, they won't continue to hold to that incorrect notion and ignore the cost of those string copies.

7. As always, the textbook and this week's section handout are chock full of great exercises and additional examples and explanations to help reinforce this material.